CS 374 Lab 2: fork( ) and exec( )

Creating programs in C/C++ that generate new processes is straightforward and is covered in chapter 11 of *Beginning Linux Programming* [*BLP* for short]; **most items needed are in pages 470-478 if you have the 4th edition from the top Module on Canvas**. Many additional helpful resources and examples are available on the web, and other readable books include *Gnu/Linux Application Programming* by Tim Jones, and *Interprocess Communications in Linux* by John Shapley Gray.

The process API is mostly represented by the <unistd.h>, the <sys/types.h>, and (if you want to print errors) the <errno.h> libraries. You will probably want <stdio.h> or <iostream.h> for output functions like printf or cout.

Key functions to familiarize yourself with in this exercise include fork(), exec(), wait(), getpid(), getppid(), and getuid(). There are many variations in the exec() call family—one of the easiest to use is execl() [see page 471 in *BLP*]. We will also use the sleep() utility to get time delays. Use the man pages for descriptions of these.

I prefer you to work in groups of 2-3 on this exercise—no more than 4 students per group. If you work with others, please make sure you take turns as to who is typing at the keyboard! I suggest you keep track of the clock and switch every 15 minutes between typist and shoulder surfer roles.

1. Your first task is to write a parent program which will fork off two new processes, where the new child processes will execute a *different* program. Part 2 of this will guide you to building the parent program. Part 3 of this will guide you through building the child program code. All three processes will sleep at least ten seconds; then print out their own process id, the process id of their parent, and their user id. While the processes are running, you will run ps to gather the information about all of your processes, then you will use that information to verify what happened and construct a process call tree.
2. Using a text editor, first create a program for the **parent process**. That program should begin by doing the forks. When it forks off the first process, it will have to catch the integer process id from the fork. Then do an if-else block to determine whether you are in the parent process or child process based on the pid value you collected. If you are the parent process, print the process id of the child process; if you are the child process, do an exec call (I recommend execl for this lab as you’ll want to specify your home directory as the location of your child process) to do a brain transplant on the child process; your last two arguments to execl should be 0’s (null pointers). You will have to create a name for the child’s executable code file (not the C source code), which you will build in the next section—don’t forget this file name.

You’re not done yet! If you are the parent you need to repeat this process a second time: fork off a second process, use an if-else block to make sure you aren’t the child process. If you are the parent, print the child process’s id; if you are the child, do the exec call. Note that you will use the same file name as before for the exec call. This parent process should also print the parent process id and the user id of the person running the program, then sleep for 10 seconds.

You’re not done yet! Double check that the parent process prints the list of messages above (process id, parent process id, and user id). Make sure your printed messages from this program are easy to distinguish from messages from the child processes! Make your parent program sleep 10 seconds after the messages. Make sure your program compiles! You can’t test it yet!

1. Now build the code file for the **child processes**. The code for the child processes is simple: wait ten seconds (use sleep!), then print out the parent process ID using getppid(), then print out your own process ID using getpid(), then exit. Make sure your print messages from the children can be distinguished from those of the parent. Compile this program making sure to declare the compiled code to be the file you named in the parent by using the –o option on the compiler. Test this program first! Make sure that the output lines are separate!
2. Now you can test the compiled version of the parent program. First just make sure it runs. Then make sure that the output lines all have carriage returns. Then do one run normally, and do a second run where you run the parent program in the background (you can do this by typing “myprog &” instead of “myprog” at the prompt). *Did the behavior change at all? What did you have to do to see output?* Note that these questions are on the Questions for the lab.
3. Draw a picture of the calling tree of the processes involved, using their file names and pid numbers from an execution run. The call tree should show **the shell** you are in, **the parent** process, and **the two child processes**.

**Example Call Tree Style (your tree will differ):**

programX (PID = 2339)

programY (PID = 2340)

programAA (PID = 2350)

programZ (PID = 2348)

1. Turn in one copy of parent code and child code for your group, along with answers to the questions on the question sheet available separately for this lab. There is a three part “quiz” to upload the three files.

// SAMPLE PROGRAMS FROM THE LAB BELOW

/\* forkdemo1.c

\* shows how fork creates two processes, distinguishable

\* by the different return values from fork()

\*/

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

main()

{

int ret\_from\_fork, mypid;

mypid = getpid(); /\* who am i? \*/

printf("Before: my pid is %d\n", mypid); /\* tell the world \*/

ret\_from\_fork = fork();

sleep(1);

printf("After: my pid is %d, fork() said %d\n",

getpid(), ret\_from\_fork);

}

/\* forkdemo2.c - shows how child processes pick up at the return

\* from fork() and can execute any code they like,

\* even fork(). **Predict number of lines of output.**

\*/

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

main()

{

printf("my pid is %d\n", getpid() );

fork();

fork();

fork();

printf("my pid is %d\n", getpid() );

}

/\* forkdemo3.c - shows how the return value from fork()

\* allows a process to determine whether

\* it is a child or process; use exec() in child

\*/

#include <stdio.h>

#include <errno.h>

#include <sys/types.h>

#include <unistd.h>

main()

{

int fork\_rv;

printf("Before: my pid is %d\n", getpid());

fork\_rv = fork(); /\* create new process \*/

if ( fork\_rv < 0 ) /\* check for error \*/

printf("An error occurred with fork (%d)\n", errno);

else if ( fork\_rv == 0 ){

printf("I am the child. My pid=%d\n", getpid());

execlp("ls", "ls", "-l", "/", (char \*)NULL);

}

else

printf("I am the parent. My child is %d\n", fork\_rv);

}